###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

# «Программирование многопоточных приложений. POSIX Threads»

студента 2 курса, 22202 группы

**Бальчинова А.С.**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

В.А. Перепёлкин

Новосибирск 2024

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc18443921)

[ЗАДАНИЕ 3](#_Toc18443922)

[ОПИСАНИЕ РАБОТЫ 3](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ 4](#_Toc18443924)

[Приложение. 5](#_Toc18443925)-8

**ЦЕЛЬ**

Освоить разработку многопоточных программ с использованием POSIX Threads API. Познакомиться с задачей динамического распределения работы между процессорами.

# ЗАДАНИЕ

Есть список неделимых заданий, каждое из которых может быть выполнено независимо от другого. Задания могут иметь различный вычислительный вес, т.е. требовать при одних и тех же вычислительных ресурсах различного времени для выполнения. Считается, что этот вес нельзя узнать, пока задание не выполнено. После того, как все задания из списка выполнены, появляется новый список заданий. Необходимо организовать параллельную обработку заданий на нескольких компьютерах. Количество заданий существенно превосходит количество процессоров.

**ОПИСАНИЕ РАБОТЫ**

В каждом процессе заведем 2 потока: вычисляющий и управляющий.

* Управляющий поток выполняет receiver;
* Вычисляющий поток выполняет executor.

Использование потоков позволяет производить перераспределение заданий на фоне счета. Благодаря этому можно добиться гораздо более эффективного использования ресурсов, чем если бы процесс должен был прерывать обработку заданий на время принятия или отсылки части работы.

# ЗАКЛЮЧЕНИЕ

В процессе выполнения лабораторной работы, мы освоили разработку многопоточных программ с использованием POSIX Threads API. Также мы познакомились с задачей динамического распределения работы между процессорами и попытались реализовать её на практике.

ПРИЛОЖЕНИЕ

#include <iostream>

#include <unistd.h>

#include <pthread.h>

#include <mpi.h>

#include <cstdlib>

#include <cstring>

#include <cmath>

#define EXECUTOR\_FINISHED\_WORK -1

#define SENDING\_TASKS 656

#define SENDING\_TASK\_COUNT 787

#define NO\_TASKS\_TO\_SHARE -565

#define ITERATION\_COUNT 10

#define TASK\_COUNT 2000

#define MIN\_TASKS\_TO\_SHARE 2

using namespace std;

pthread\_t threads[2];

pthread\_mutex\_t mutex;

int\* tasks;

double summaryDisbalance = 0;

bool isFinishedExecution = false;

int num\_of\_proc;

int procrank;

int remainingTasks;

int executedTasks;

int additionalTasks;

double globalRes = 0;

void initTasks(int \*taskSet, int taskCount, int iterCount) {

    for (int i = 0; i < taskCount; i++) {

        taskSet[i] = abs(procrank - (iterCount % num\_of\_proc));

    }

}

void executeTasks(const int\* taskSet) {

    pthread\_mutex\_lock(&mutex);

    int startCount = remainingTasks;

    pthread\_mutex\_unlock(&mutex);

    for (int i = 0; i < startCount; i++) {

pthread\_mutex\_lock(&mutex);

        if (startCount >= remainingTasks && i >= remainingTasks) {

pthread\_mutex\_unlock(&mutex);

            break;

        }

        int weight = taskSet[i];

        pthread\_mutex\_unlock(&mutex);

        for (int j = 0; j < weight; j++) {

            globalRes += sqrt(j);

        }

        executedTasks++;

    }

    remainingTasks = 0;

}

void\* executor(void\* args) {

    tasks = new int[TASK\_COUNT];

    double startTime, finishTime, iterationDuration, shortestIteration, longestIteration;

    for (int i = 0; i < ITERATION\_COUNT; i++) {

        startTime = MPI\_Wtime();

        MPI\_Barrier(MPI\_COMM\_WORLD);

        executedTasks = 0;

        remainingTasks = TASK\_COUNT;

        additionalTasks = 0;

        executeTasks(tasks);

        int threadResponse;

        for (int procIdx = 0; procIdx < num\_of\_proc; procIdx++) {

            if (procIdx != procrank) {

                MPI\_Send(&procrank, 1, MPI\_INT, procIdx, 888, MPI\_COMM\_WORLD);

                MPI\_Recv(&threadResponse, 1, MPI\_INT, procIdx, SENDING\_TASK\_COUNT, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

                if (threadResponse != NO\_TASKS\_TO\_SHARE) {

                    additionalTasks = threadResponse;

                    memset(tasks, 0, TASK\_COUNT);

                    MPI\_Recv(tasks, additionalTasks, MPI\_INT, procIdx, SENDING\_TASKS, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

                    pthread\_mutex\_lock(&mutex);

                    remainingTasks = additionalTasks;

                    pthread\_mutex\_unlock(&mutex);

                    executeTasks(tasks);

                }

            }

        }

        finishTime = MPI\_Wtime();

        iterationDuration = finishTime - startTime;

        MPI\_Allreduce(&iterationDuration, &longestIteration, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

        MPI\_Allreduce(&iterationDuration, &shortestIteration, 1, MPI\_DOUBLE, MPI\_MIN, MPI\_COMM\_WORLD);

        MPI\_Barrier(MPI\_COMM\_WORLD);

        summaryDisbalance += (longestIteration - shortestIteration)/longestIteration;

    }

    pthread\_mutex\_lock(&mutex);

    isFinishedExecution = true;

    pthread\_mutex\_unlock(&mutex);

    int signal = EXECUTOR\_FINISHED\_WORK;

    MPI\_Send(&signal, 1, MPI\_INT, procrank, 888, MPI\_COMM\_WORLD);

    delete[] tasks;

    pthread\_exit(nullptr);

}

void\* receiver(void\* args) {

    int askingProcRank, answer, pendingMessage;

    MPI\_Status status;

    MPI\_Barrier(MPI\_COMM\_WORLD);

    while (!isFinishedExecution) {

        MPI\_Recv(&pendingMessage, 1, MPI\_INT, MPI\_ANY\_SOURCE, 888, MPI\_COMM\_WORLD, &status);

        askingProcRank = pendingMessage;

        if (remainingTasks >= MIN\_TASKS\_TO\_SHARE) {

            pthread\_mutex\_lock(&mutex);

            answer = remainingTasks / (num\_of\_proc \* 2);

            remainingTasks = remainingTasks / (num\_of\_proc \* 2);

            pthread\_mutex\_unlock(&mutex);

            MPI\_Send(&answer, 1, MPI\_INT, askingProcRank, SENDING\_TASK\_COUNT, MPI\_COMM\_WORLD);

            MPI\_Send(&tasks[TASK\_COUNT - answer], answer, MPI\_INT, askingProcRank, SENDING\_TASKS, MPI\_COMM\_WORLD);

        }

        else {

            answer = NO\_TASKS\_TO\_SHARE;

            MPI\_Send(&answer, 1, MPI\_INT, askingProcRank, SENDING\_TASK\_COUNT, MPI\_COMM\_WORLD);

        }

    }

    pthread\_exit(nullptr);

}

int main(int argc, char\* argv[]) {

    int threadSupport;

    MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE, &threadSupport);

    if (threadSupport != MPI\_THREAD\_MULTIPLE) {

        MPI\_Finalize();

        return -1;

    }

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &procrank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_of\_proc);

    pthread\_mutex\_init(&mutex, nullptr);

    pthread\_attr\_t threadAttributes;

    pthread\_attr\_init(&threadAttributes);

    pthread\_attr\_setdetachstate(&threadAttributes, PTHREAD\_CREATE\_JOINABLE);

    double start = MPI\_Wtime();

    pthread\_create(&threads[0], &threadAttributes, receiver, nullptr);

    pthread\_create(&threads[1], &threadAttributes, executor, nullptr);

    pthread\_join(threads[0], nullptr);

    pthread\_join(threads[1], nullptr);

    pthread\_attr\_destroy(&threadAttributes);

    pthread\_mutex\_destroy(&mutex);

    if (procrank == 0) {

        cout << "Proc " << procrank << ": Summary disbalance:" << summaryDisbalance / ITERATION\_COUNT \* 100 << "%" << endl;

        cout << "Proc " << procrank << ": time taken: " << MPI\_Wtime() - start << endl;

    }

    MPI\_Finalize();

    return 0;

}